APPLICATION FOR UNITED STATES LETTERS PATENT

by

MATTHEW N. SCHMID

JOHN THOMAS BLOCH

FRANK F. HILL

and

ANUP K. GHOSH

for

SYSTEM AND METHOD FOR DEFENDING AGAINST MALICIOUS SOFTWARE

SHAW PITTMAN
1650 Tysons Boulevard
McLean, VA 22102-4859
(703) 770-7900
Attorney Docket No.: CIG-101

SYSTEM AND METHOD FOR DEFENDING AGAINST MALICIOUS SOFTWARE

[0001] This application claims the benefit of U.S. Provisional Application No. 60/193,397, filed March 31, 2000.

BACKGROUND

Field of the Invention

[0002] The present invention relates to the field of computer security, and more specifically relates to controlling the software which may be run on a computer.

Background of the Invention

[0003] The distribution of malicious software has plagued computer networks since the days of electronic bulletin board systems (BBSs). While these publicly accessible sites allowed users to exchange software, it was well understood that untrusted software could be malicious. As a rule of thumb software downloaded from a BBS was not installed on essential computer systems unless it was somehow determined to be safe.

[0004] Today the Internet dominates computing, and because connectivity and interoperability are essential systemic qualities, malicious software has become an even more pernicious problem. Web sites provide mobile code, such as ActiveX controls and signed Java applets, which can be seamlessly, and often transparently, installed on a computer and executed through a common Web browser. While such software may enhance the multimedia experience, create a more interactive environment, or provide other positive benefits, the same technology allows software to be written which locates and destroys documents. Documents accessible to such

software include not only those stored on the computer on which the software is installed, but also those available across the network to which the computer belongs.

[0005] In addition to Web browser based malicious software, users also run programs received via E-mail for entertainment, without knowing what the program will do, or its origin in the world. Other executables can masquerade as data, donning, for example, DOC, GIF, and JPEG file extensions, and causing their own type of damage.

[0006] A study by Computer Economics of Carlsbad, California, found that in the first two quarters of 1999 alone, businesses worldwide lost more than $7.6 billion to malicious software. Additionally, well-publicized events involving the Melissa virus, the Explorer.zip worm, and BackOrifice clients contribute to an atmosphere of crisis. Organizations may respond to these threats by adopting policies prohibiting software download or installation, but, in practice, organizations have little control over this activity.

[0007] The idea of restricting application execution has been explored by some in the prior art, but none adequately addresses issues of strong security and easy administration. Some in the prior art allow administrators to set up rules for local computers, which define times during which programs can be executed, and by whom such programs may be executed. In addition, the prior art has introduced the concept of rejecting unknown applications.

[0008] However, those in the prior art have several weaknesses which render them inadequate. For example, those rejecting unknown applications base the determination of which applications are known solely on the application's executable

name. This system allows even a novice user to bypass the security system by naming a malicious application after an approved one. In addition, many in the prior art provide inadequate administrative support over a user community. For example, the prior art only allows administration on a per-machine basis, and policies must be defined explicitly for each user.

[0009] Others in the prior art have taken a different approach to control of executable software. In those products, cryptographic hashes of critical files are maintained and periodically verified. An application on the computer periodically checks the critical files, and if changes are detected, an administrator may be notified. This technology is used primarily for intrusion detection. However, execution control is not combined with error checking, thus an executable that has been compromised by a virus (and has therefore had its signature altered) will be allowed to execute until the system has detected this change and an administrator has removed the compromised executable. A significant amount of damage could be done before the problem is detected.

SUMMARY OF THE INVENTION

[0010] The present invention improves upon prior malicious software protection applications by not only resolving the problems discussed above, but also by addressing emerging dangers and unknown attacks. The present invention includes an execution management utility designed to prevent software from executing without the prior approval of system administrative staff. In a preferred embodiment, a kernel module, loaded by Windows NT systems, may be capable of selectively intercepting process creation requests.

[0011] Although the present invention is inspired by malicious software, the present invention also addresses other problems. For example, the present invention can assist corporations by enforcing policies regarding unauthorized, unlicensed, or pirated software, such as, but not limited to, games; entertainment software; and non-standard utilities, such as advertising-enhanced browsers.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012] Figure 1 is a block diagram illustrating communication between an EMU Client, operating in user-space, and an Execution Manager, functioning as a device driver in kernel-space.

[0013] Figure 2 is a flow diagram illustrating a method through which a device driver may be unwrapped without warning.

[0014] Figure 3 is a block diagram illustrating a kernel-level wrapping technique.

DETAILED DESCRIPTION OF THE INVENTION

[0015] The present invention improves upon existing executable control software, and provides administrators with a valuable defense against the introduction of hostile or unknown code. The present invention is comprised of an execution management utility (EMU), which restricts a user to executing only an approved and known set of applications. Such applications typically include application-specific operating system software, operating system services, and a set of applications necessary for a user to perform his or her duties. In addition, an EMU provides a client-server architecture that enables an administrator to restrict applications available to individual users working on a workstation with an EMU client installed. This allows administrators to address specific user software needs.

[0016] To properly implement a security system based on an EMU, an "EMU client" may be required. An EMU client may comprise a service, or other operating system level component, that runs on each protected computer. Some operating systems provide security necessary to ensure that the service will run, and that only an administrative user can shut it down.

[0017] At the core of an EMU is an Execution Control List, or ECL. In a preferred embodiment, each user on the system has his or her own ECL. Each ECL may contain a table of applications a user is authorized to run and corresponding cryptographic codes that uniquely identify each application. In a preferred embodiment, the present invention utilizes the MD5 cryptographic algorithm, as defined in Internet RFC 1321, to create the cryptographic identifier, or "hash."

[0018] Identifying a program through its hash reduces the likelihood that a user can bypass the EMU simply by changing the name of an unknown executable to the name of an executable that is already part of an ECL. In addition, identifying a program through its hash may reduce the potential impact of viruses; many viruses propagate by silently attaching themselves to an application and executing whenever the application is run.

[0019] Using hashes for identification has the additional benefit of preventing modified applications from executing. An EMU may ensure that modified executables will not run, effectively protecting a system from any damage these altered applications could cause.

[0020] An EMU client may intercept execution requests and verify whether a requesting user has permission to run a requested application. In a preferred

embodiment, an EMU client may verify an execution request by creating an MD5 cryptographic hash of a requested executable, and then looking for this name/hash combination in a user ECL. In this embodiment, a name is used in addition to the hash to ensure that a user is executing the program he or she intended to execute. For example, the name/hash verification would prevent an attack in which a program that deletes files is accidentally renamed to that of one that compresses files for archival (assuming that both of these are in the user's ECL). If a name/hash combination were not used, execution of the aforementioned program would result in the deletion of files.

[0021] In one contemplated embodiment, the present invention may utilize locally stored and locally maintained ECL's. However, in this embodiment, each EMU client would be required to maintain a copy of all ECL's for every user on the system. Without such copies, users would not be able to easily move from computer to computer and have their ECL's follow them. In a preferred embodiment, user mobility may be facilitated through an Administrative Server ("AS").

[0022] An AS may facilitate user mobility by storing master copies of all ECL's in a common location, from which an EMU client may download an appropriate ECL when a new user accesses the computer. In addition to ECL's for each user, an AS may contain a default ECL which may be locally cached by an EMU client. A default ECL may be used when an EMU client does not have an ECL for a requesting user and cannot obtain an ECL for a user from an AS. Providing a default ECL allows a user authorized to log onto an EMU-protected workstation to be provided with a basic, pre-approved ECL.

[0023] To make itself useful on an enterprise level, an AS may also provide an interface by which all administrable EMU clients may be accessed through a centralized location. An administrator can manage policies and control EMU client functionality from a single machine running as an EMU client's AS. An administrator may also control ECL distribution to EMU clients through an AS.

[0024] For example, an AS may allow an administrator to send a command to a specific EMU client, or a group of EMU clients, which forces an ECL update. This might be beneficial after an administrator has made changes to an ECL, and wants a client to implement the new ECL.

[0025] In addition to server-required updates, an EMU client may request an ECL from an AS under several different circumstances. For example, if an EMU client does not have a local copy of an ECL for a requesting user, an ECL may be requested. In addition, an EMU client may store and refer to a time-stamped local copy of a user's ECL. After an administrator-defined period, an EMU client may request an updated copy of an ECL.

[0026] An AS may process user ECL requests and distribute appropriate ECL's to EMU clients via the local area network. Each ECL maintained by an AS contains a hash of ECL contents. When an EMU client receives a user ECL from an AS, ECL signature verification may occur. If a signature is valid (indicating that an ECL originated from an AS and has not been modified), an ECL may be accepted for use by an EMU client. An invalid signature may result in ECL rejection.

[0027] In addition to allowing an administrator to monitor and modify ECL's, an AS may also handle user requests to add applications to an ECL. A user can generate

such a request when an EMU client rejects execution of an application that is not on a user ECL. When an EMU client prevents an application from executing, an EMU client may generate a dialog box indicating that the requested executable is not on a current ECL, and may ask a user if she would like to request the addition of the requested executable to her ECL. If she opts to submit a request, an ECL Add Request is forwarded to an AS. If an AS is unavailable, an execution request may be rejected.

[0028] An add request may consist of a user name, program name, program hash, and other such information. An AS may maintain a queue of add requests awaiting approval or rejection by an administrator. To make an administrator's job easier, an AS may maintain a database of known application hashes. If an add request is received for a known application, an AS may inform an administrator of this positive identification. If a program's hash is unknown, an administrator may investigate the application before allowing its addition to a user ECL. If an administrator chooses to accept an add request, a user ECL is updated and a workstation requesting an ECL change may be signaled to update its ECL.

[0029] In addition to user- and EMU client-generated requests, an AS may also allow executables to be added to an ECL through an ECL editor, and through a "learning mode. In a preferred embodiment, an ECL may be developed using an ECL editor. An ECL editor allows an administrator to select executable files to be added to user ECL's. An administrator may also use an ECL editor to modify existing policies. Additionally, an ECL editor may allow an administrator to make changes to groups of policies all at once. This functionality may be useful when, for example, an

administrator decides that all users in a group should be able to execute certain applications. However, developing an entire policy using an ECL editor may be time consuming, so an EMU may provide other methods for policy development.

[0030] In a preferred embodiment, an EMU client may include a learning mode, which may enable an EMU client to develop an ECL policy customized for individual users. Learning mode can be enabled on either a per-user or per-machine basis. An administrator can enter a password into an EMU client to set a machine in learning mode. A special entry in the user's ECL can also be used to signify that a user is in learning mode. Once in learning mode, an ECL can be generated by observing user application usage patterns. During learning mode, an EMU passively observes program execution and does not attempt to enforce any ECL restrictions. Each application that a user executes may be added to his or her current ECL.

[0031] Developing custom policies through learning mode enables an administrator to limit a user to those programs necessary and sufficient to perform his or her job. These functions may include operating system utilities and services that start up on machine boot. After allowing a user to work under learning mode for a specified period of time, an administrator may review a generated ECL and make any desired changes.

[0032] A process-based tracing facility may provide all of the above described features. However, in the presently preferred Windows NT embodiment, there are several locations where a process-based tracing facility can be implemented. In the user space, a process tracing facility can be implemented between Windows applications and the Win32 subsystem. This is the location where most Windows

wrappers are implemented. The benefit of this approach is that all code can be written in user mode and the kernel need not be modified. The drawback of this approach, which renders it infeasible for protection from malicious software, is that it is easily bypassed by making direct calls to the operating system.

[0033] While most well-behaved Windows applications make system calls through the Win32 API, it is also possible to bypass this interface by making calls directly to kernel services. Nothing in the Windows NT kernel prevents making direct calls to executive objects. A malicious program cognizant of Windows wrapping approaches may attempt to bypass a wrapper implemented between Windows and the Win32 subsystem by making direct calls to operating system objects, rather than through the Win32 API.

[0034] To provide proper integration with an operating system, the present invention utilizes an kernel-level wrapping approach to executable control. To implement a non-bypassable kernel wrapper, implementation should take place within the kernel. Installing a device driver is a method for adding user-written kernel-mode code to the operating system kernel. Writing a device driver provides access to internal operating system functions and data structures not accessible from user mode. Device drivers may be installed in an I/O subsystem, and may be used to intercept system service requests.

[0035] System services are operations performed by an operating system kernel on behalf of applications or other kernel components. These operations are implemented as system services because they may affect processes other than a calling process. For instance, manipulating CPU hardware, starting and stopping processes, and

manipulating files are sensitive operations generally implemented by services at the kernel level.

[0036] In a preferred Windows NT embodiment, user applications invoke system services by executing an interrupt instruction. Kernel code temporarily takes control of a computer in response to an interrupt, and often performs some useful activity for a calling process before relinquishing control. A kernel entity, known as a dispatcher, initially responds to an interrupt instruction, determines the nature of an interrupt, and calls a function to handle the request. Two tables in kernel memory describe locations and parameter requirements of all functions available to a dispatcher. One table specifies handlers for user requests; the other specifies handlers for requests originating within the kernel. The calling process places information about the requested system service on the stack along with any parameters required to complete the operation.

[0037] The present invention may control new process creation by instructing a dispatcher to call a specific function when a user process invokes certain system services. This approach requires constructing a device driver that may be dynamically loaded into a kernel, or may be loaded as part of a boot sequence. When a device driver loads, it modifies an entry in a table consulted by a dispatcher to handle an interrupt instruction. In the preferred Windows NT embodiment, the modified entry may cause a dispatcher to call an inserted function rather than ZwCreateProcess, the NT function responsible for creating new user- and kernel-mode processes. It is important to note that the driver modifies only tables relevant to requests from user applications. This is because we are not interested in controlling

the kernel's ability to create processes. In addition, the inserted wrapper function exposes properties and methods similar to those of ZwCreateProcess, thereby limiting the impact of the kernel wrapper.

[0038] When a substitute function is called by a dispatcher, the function determines whether to allow or block process creation. If the substitute function wishes to allow process creation, ZwCreateProcess may be invoked with a user's parameters, and results may be passed to a calling function. If process creation is not allowed, a failure value may be returned. There are several possible ways by which a substitute function may evaluate a creation request.

[0039] In a preferred embodiment, a wrapping function may contact a user-mode application to obtain a process creation request ruling. The wrapping function may provide information about requesting and requested processes (i.e. the current process and the process it wishes to create). A user-mode application may engage its own logic, and inform the wrapping function of its decision. This logic may include consulting a list of approved executables, prompting for permission to proceed, or requiring more sophisticated authorization (such as, but not limited to, a password).

[0040] Note that once a wrapping function intercepts an execution request, a decision must be made about the fate of the request. As discussed above, a wrapping function contacts a user-mode application running as a service (not to be confused with a system service), provides information about a request, and receives instructions to permit or deny creation of a new process.

[0041] A kernel wrapping function may communicate with a user-mode service. An I/O Subsystem may provide a communication mechanism which allows applications

to pass messages and data to device drivers. In a preferred embodiment, messages may be represented as control codes, called IOCTLS, and may be passed to device drivers, with or without data, via an API function, such as DeviceIoControl. However, such an API function may not provide an adequate communication channel because of its asymmetric nature. Communication via DeviceIoControl may be initiated by the application; a device driver merely responds to requests.

[0042] In a preferred embodiment, a wrapper may notify a service when assistance is needed, instead of waiting for an inquiry from a service. When DeviceIoControl calls or other, similar calls, are paired with synchronization objects, two-way communication between a service and a kernel may be established. Using this approach, a device driver may indicate an interception by releasing a semaphore. This may cause a service thread to resume execution and to call DeviceIoControl, thereby retrieving information about an execution request from a kernel. After a service considers input from a kernel, a service may return a verdict via DeviceIoControl.

[0043] Alternative, polling-based embodiments are also contemplated. However, polling-based embodiments have proven less efficient than the embodiment described above. Another embodiment contemplated employed API-level wrapping, rather than kernel-level wrapping, in a effort to simplify communication. However, this API-level wrapping raises a specter of multithreaded, race condition attacks, where one thread applies for and receives permission to execute an innocuous application, such as clock.exe, and another thread replaces clock.exe with something threatening, such as napster.exe. In this example, a data swap may take place after checks on clock.exe

is validated but before a call to CreateProcess. This is an example of a time of check versus time of use vulnerability that permits race condition attacks.

[0044] From both a security and a programmability perspective, bi-directional communication between device drivers and user-mode applications may be more appealing when details are hidden by a layer of abstraction. As illustrated in Figure 1, a preferred embodiment of the present invention may employ a general-purpose library which provides communication facilities mimicking pipe or socket communication. Kernel entities and applications may use “accept” and “close” functions to create and destroy connections; “read” and “write” calls extract data from or insert data into the “pipe” abstraction provided by a library. A library may hide data buffering and synchronization details and facilitate rapid development and thorough testing.

[0045] Pipe- or socket-like communication may be facilitated by enhancing a kernel-level function by wrapping a more advanced function around the kernel-level function. A simple method of wrapping involves making a local copy of relevant system service entries and placing a wrapper function in place of those service entries. This method also works well when multiple people install wrappers; in effect, a function chain is created, in which a function thinks that it is calling an appropriate System Service. However, as illustrated by Figure 2, this chain can easily become disrupted if device drivers begin unwrapping system services.

[0046] In addition to enhancing communications, it is also desirable for a wrapping function to load and unload dynamically. To achieve this goal, the present invention addresses the difficulties presented when unloading a device driver which has

wrapped a system service. Specifically, because a user mode program may call a system service at any time, it can be difficult to tell when a wrapper is in use. A driver may not be unloaded safely until a wrapper is no longer in use.

[0047] To perform service wrapping properly, a wrapper should have at least two important features: the ability to allow multiple, concurrent wrappers, and a framework under which drivers can safely wrap and unwrap system services. Although there are several possible solutions to allowing multiple concurrent device drivers, a preferred embodiment exports functionality to the entire kernel. This approach may allow other device drivers to call exported functions that add to, remove from, and descend through layers of functions on top of existing system services. This approach has the added benefit of hiding implementation details, and allowing a wrapper to carefully synchronize access to layers surrounding a system service table.

[0048] The synchronization of system service access functions allows the present invention to provide safe wrapping and unwrapping of system services. In a preferred embodiment, actions that add or remove wrapper layers are considered "writers," while actual calls to system services that descend through the layers are considered "readers." Consistent with traditional synchronization problems, multiple readers are allowed to descend through wrapper layers simultaneously, but writer actions, which may modify wrapper layers, should be serialized. As illustrated by Figure 3, tightly controlling reader and writer synchronization allows a device driver to safely unload after removing all of its system service table layers.

[0049] An EMU designed as described above may provide security-conscious administrators with a valuable defense against the introduction of new and potentially malicious code. Such an EMU ensures that only those applications explicitly approved by an administrator may be executed by users. This greatly reduces the threat posed by viruses, Trojan horses, and even malicious insiders. In addition to security considerations, an EMU also provides control over illegally-obtained application distribution and entertainment program use.

[0050] As previously described, a preferred EMU embodiment includes numerous features that make it manageable at the enterprise level, including centrally-managed execution control lists and client-server communication. A kernel based wrapping approach ensures the non-bypassability of an EMU and may result in negligible performance overhead. Security features incorporated in an EMU ensure that even a malicious adversary may not circumvent an ECL.

[0051] In today's Internet-enabled work environment, it is essential that security administrators have full control over the programs that are executed by their user community. The EMU's ability to control which programs are allowed to execute is a significant step toward maintaining a secure networked environment.

[0052] While the preferred embodiment and various alternative embodiments of the invention have been disclosed and described in detail herein, it may be apparent to those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope thereof.